University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Automatic Generation of ROP-based Payloads

Thesis Supervisor:
Ș.l.dr.ing Răzvan Deaconescu
razvan.deaconescu@cs.pub.ro

Author:
Alexandru Răzvan Căciulescu
alexandru.razvan.c@gmail.com

Bucharest, 2016

Special thanks to Sergi Alvarez (pancake),
Anton Kochkov (xvilka) and Jeff Crowell
for providing great assistance
with the Radare framework.

Abstract

Modern architectures are now enforcing Data Execution Prevention mechanisms which prevent applications or services from executing code from non-executable memory regions. Return oriented programming is a technique that allows the hijacking of an executable even in the presence of such security defenses. Using this technique and building ROP chains by hand can be a tedious and error-prone process. This project aims to provide a fast and intuitive tool to aid in this process and automate it.

Contents

List of Figures

List of Tables

List of Algorithms

Notations and Abbreviations

CLI – Command Line Interface
CTF – Capture The Flag
DEP – Data Execution Prevention
ESIL – Evaluable String Intermediate Language
FLG – Flag
IR – Intermediate Representation
ISA – Instruction Set Architecture
MEM – Memory
PR – Pull Request
REG – Register
ret2libc – Return to libc
ROP – Return Oriented Programming
SDB – String Data Base
SMT – Satisfiability Modulo Theories
VM – Virtual Machine
W^X – Write xor Execute

Chapter 1

Introduction

1.1 Project Description

Since it was first introduced in 2007, Return Oriented Programming[14] (and derived techniques) proved to be a powerful technique that allows the hijacking of an executable control flow even in the presence of modern defenses such as Data Execution Prevention, which are present on modern architectures. It can be seen as a generalization of the return-into-libc attack, which instead of calling functions relies on short instruction sequences, called gadgets, to allow arbitrary computation. This in turn creates a resilient technique immune to defense mechanisms such as removing certain functions from libc or changing the assembler code generation process.

1.2 Project Motivation

The process of hijacking using ROP consists of building a hand-crafted payload that chains together several gadgets in order to overwrite and hijack the control flow of the executable program with a specific goal in mind. The user has to determine which gadgets to use and chain them together while keeping in mind how these may affect the overall state of the program. However this can be a tedious and error-prone process.

1.3 Project Objectives

In this paper I introduce a new intuitive tool with the objective to aid and automate the classification of gadgets, selection and creation of the final ROP payload which the user can later feed to the executable. I decided to build and integrate this tool directly with an already existing open-source reverse engineering framework, Radare2.

Radare2 was the ideal choice since it is a mature framework with an impressive set of tools and features, among these we can count its own custom intermediate language representation as well as its internal virtual machine used for emulation. These represent the basic building blocks on which this project will be based.

In the following chapters I will present the basic concepts and techniques used for ROP as well as an overall presentation of the Radare2 framework and how my project integrates with it.

Chapter 2

Basic Concepts

Before proceeding forward it is necessary to understand some of the basic concepts behind hijacking the control flow of an executable and how they evolved into Return Oriented Programming.

2.1 Terminology

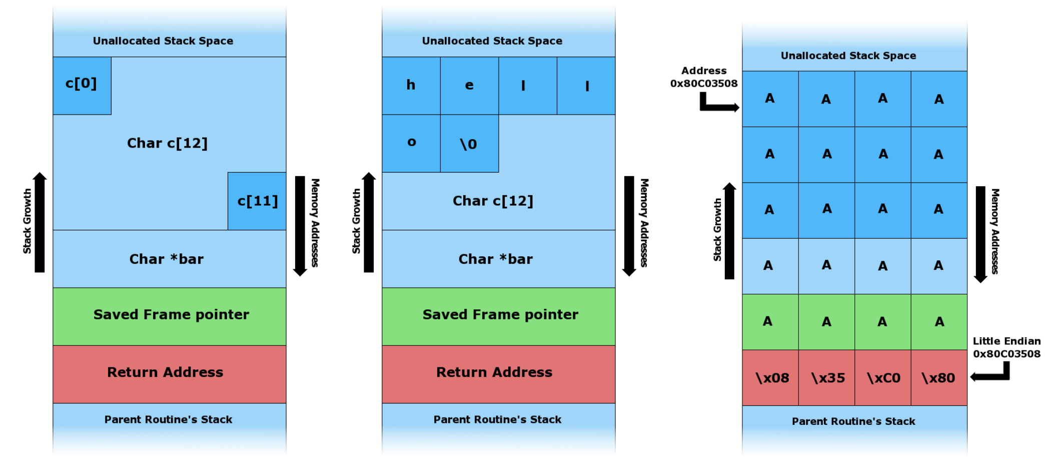
Throughout this paper I will refer to the creation of the ROP **payload** and injection through an **exploit** / **vulnerability**. To better understand the concept let's consider the following analogy: at a customs office someone is trying to smuggle in contraband through a secret passage. The payload represents the contraband we are trying to introduce and the exploit represents the method of delivery, in this case the secret passage. If there is no contraband the secret passage is useless by itself, similarly the contraband is of no use if without a method to smuggle it in.

Other common terms used in the security world are **attack vector** and **attack surface**. The former represents the path or means an attacker uses to reach his target, in this context the attack vector is represented by the Return Oriented Programming technique. The latter represents the sum of all attack vectors present which can be used by the attacker.

2.2 Stack Smashing Attack

One of the traditional exploits[6] used by attackers was to try and manipulate the call stack by taking advantage of a bug in the program, usually a buffer overflow. This happens when a function does not perform proper bounds checking and accepts more input data than it can store properly. The excess data can overflow on the stack and overwrite the return address (Figure 2.1), which can be used to hijack the control flow. The first step of the attack consists of creating a malicious payload with the intent to run arbitrary code, followed by overwriting the return address of a function call to point to the previously injected code.

The buffer overflow exploit has been a favorite technique for attackers over the years and proved to be quite versatile[5].



(a) Initial Buffer (b) No Buffer Overflow (c) Buffer Overflow

Figure 2.1: Stack Smashing Example[1]

2.3 Data Execution Prevention

In order to prevent attackers from exploiting buffer overflows with this simple technique, modern architectures now implement a Data Execution Prevention[10] (often referred to as DEP or W^X) mechanism which marks the writable regions of memory as non-executable. This guarantees that no region in the memory can be both executable and writable at the same time. Even if an attacker manages to inject a payload and overwrite the return address with the payload location, when the code will jump and try to execute the malicious code it will cause an exception and the program will crash.

2.4 Return to libc

Because of the DEP mechanism the attacker is now restricted to code already marked as executable from the memory. Since shared libraries, such as libc, contain subroutines for system calls and other potentially useful functions for an attack, instead of writing a payload onto the stack, the target will now be to overwrite the return address with the entry location of a library function.

Even with this technique[9][4] it is difficult to mount a successful attack since most libraries began to remove or restrict functions that would potentially be useful for an attack as a security measure.

[1] https://en.wikipedia.org/wiki/Stack_buffer_overflow

2.5 Return Oriented Programming

Return Oriented Programming can be see as an enhanced ret2libc attack, with the generalization that instead of using whole functions it only uses short assembly instruction sequences already in memory, called gadgets. Each gadget normally executes a very specific and simple task, such as moving a register or incrementing a value for example. In order to chain these gadgets together and create a payload it is ideal to find those gadgets which end in a return instruction (i.e. ret on x86), this will make the instruction pointer jump to the next address on the stack, thus chaining the gadgets together. Figure 2.2 illustrates how the stack looks during a ROP attack, each gadget on the stack representing an address for jumping to a specific place in memory to execute the small portion of code (the actual gadget) before jumping back onto the stack to the next gadget address.

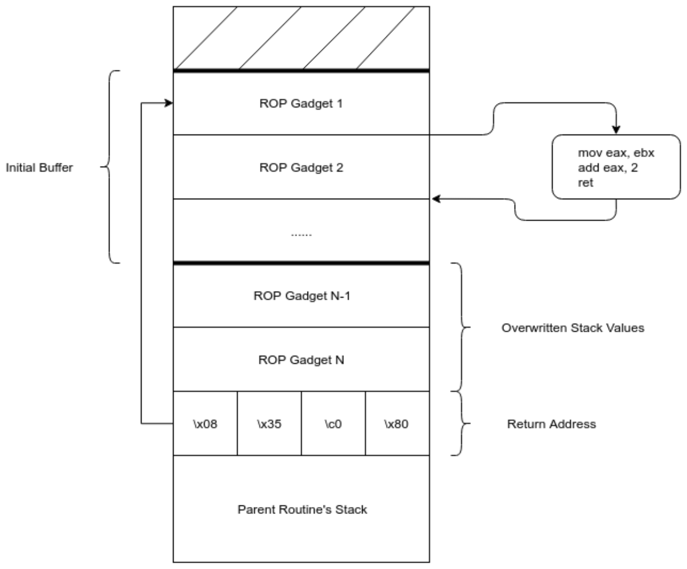


Figure 2.2: Stack During ROP Attack

Using gadgets proves more resilient to security measures because no matter what functions or calls are removed or restricted from a library, or any large body of executable code, it will inevitably contain useful gadgets which can be used to mount an attack. By using gadgets as the building blocks for this technique it also gains several advantages over the traditional ret2libc:

- Provided enough gadgets, it is Turing-complete
- Not limited to function calls

The applicability and power of the technique has been proven on different architectures[11] along with the definition of efficient algorithms for analyzing and discovering ROP gadgets.

Currently this technique requires building hand-crafted ROP chains, which can be a tedious and error-prone process. Furthermore the user must have a deep understanding of low level programming as well as the specific architecture on which the executable runs.

Chapter 3

Related Work

Although there are several tools that provide a working proof of concept for ROP attacks, there is currently no tool which automates the ROP attack to its full Turing complete potential without additional effort. Those that come close require a great deal of user input or specific knowledge of the targeted program and architecture.

3.1 ROPgadget

A de facto tool which automates the process by locating gadgets and constructing a ROP attack against a binary is ROPgadget[12]. Currently this tool supports different formats such as ELF/PE/Mach-O on different architectures x86, ARM, MIPS etc.

It's main advantage consists in its simplicity, the user only has to feed the executable and it will generate a Python code for the payload, if it finds the specific gadgets it needs.

However it has a major drawback, it is limited in the sense that it will only generate payloads which spawn a shell. If the user wants to mount an attack with any other goal than spawning a shell this is not the tool to use. There is no method to customize the end result of the attack.

To the best of my knowledge there is currently no full-fledged tool or framework which incorporates a fully customizable and intuitive tool for mounting a ROP attack without previous expertise and manual input. During my research for this project I learned of only one other project which implements something similar.

3.2 Q: Exploit Hardening Made Easy

In 2011, *Schwartz, Avgerinos and Brumley* introduced for the first time in their paper[13] the idea of semantically classifying a gadget based on it's effects rather than using the traditional approach of pattern matching. For example, rather than defining only patterns like *mov r1, r2; ret;* as a move gadget, using a semantic classification we can find equivalent gadgets such

Type	Semantic Definition
NOP	No changes in registers or memory
Jump	EIP ← Address + Offset
MoveReg	OutReg ← InReg
LoadCOnst	OutReg ← Value
Arithmetic	OutReg ← InReg1 <op> InReg2
ArithmeticConst	OutReg ← InReg1 <op> Value
LoadMem	OutReg ←[AddrReg + Offset]
StoreMem	[AddrReg + Offset] ← InReg
ArithmeticLoad	OutReg ← OutReg <op> [AddrReg + Offset]
ArithmeticStore	[AddrReg + Offset] ← [AddrReg + Offset] <op> InReg

Table 3.1: Gadget Types

as *imul r1, r2, 1; ret;* which don't contain the mov instruction at all. This offers a significant improvement and helps overcome the situations when certain types of gadget functionality cannot be found using only pattern matching.

The paper[13] defines a new instruction set architecture consisting of 9 gadget types on which this project is based as well. Rather than using pattern matching or other similar techniques for classifying the gadgets, the meaning of each gadget type is specified using a postcondition which must be found true after executing it. The complete ISA with its semantic definition can be seen in Table 3.1.

3.3 BARF

BARF[2] is a Python package "Binary Analysis and Reverse Engineering Framework" which supports only a few features compared to Radare2 or IDA[1]. The framework currently supports only a subset of the x86 and ARM instruction set, discarding any instruction which is not supported.

Among other tools, it contains BARFgadgets: a tool for classifying and verifying ROP gadgets inside a binary using SMT solvers. In the initial phases it searches for all gadgets ending in a: ret, jmp or call instruction. The next phases classifies the gadgets into predetermined types, followed by a verification phase which uses an SMT solver to verify the semantic of each gadget.

[1] https://www.hex-rays.com/products/ida/

Chapter 4

Motivation and Objectives

4.1 Motivation

Due to the high potential of this method and complex mechanisms a proper user-friendly tool is needed. This will aid in the otherwise tedious process of manually creating a ROP attack, while also providing full customization of the end result.

Currently, to our knowledge, no such tool exists. The ones that do exist do not provide full customization of the end result or are difficult to use and require specific knowledge for the method or use case in question.

4.1.1 Complexity of Mounting an Attack

Firstly, the process of searching for gadgets and deciding which ones to use in particular can be a difficult and error-prone process even for the experienced user. Depending on the desired outcome of the attack the use of a specific gadget can have dire consequences on the overall state of the attack, mostly because gadgets tend to have side effects.

Usually gadgets incorporate more than the useful instruction(s) we would like to use, thus clobbering other registers in the process. This can be at times unavoidable and it is imperative that the user takes the side effects into account, as they can divert the control flow from the desired course, forcing the user to introduce additional gadgets just for correction.

Table 4.1 illustrates a few examples of gadgets with their semantic interpretation and side effects. We can see for example that a gadget such as *add eax, 0x2; mov ebx, ecx; ret;* can be either used for incrementing eax with 0x2 which would clobber the ebx register or it could be used to move ecx in ebx, thus clobbering the eax registers. This shows how most of the time we use a gadget only for a subset of its instructions with the price of its side effects.

Ideally each gadget should be classified by their effects on the overall state of the registers and memory. This would provide a significant boost in quality and speed for the creation of the

Gadget	Semantic Interpretation	Clobbered Registers
mov al, bl ret	al ← bl	eax
add eax, 0x2 mov ebx, ecx ret	eax ← eax + 2	ebx
add eax, 0x2 mov ebx, ecx ret	ebx ← ecx	eax
pop ebp pop edx ret	ebp ← MEM[esp]	esp, edx
pop ebp pop edx ret	edx ← MEM[esp + 0x4]	esp, ebp

Table 4.1: Gadget Side Effects

payload.

Secondly, it is required to have a deep understanding of assembly and the specific architecture on which the attack is based. The same ROP payload will not work for the same program compiled on a different architecture. For a different architecture the whole process will have to be redone from the beginning, taking into account the differences for the architecture and assembly language. This poses a major drawback with most tools since they are not architecture-independent.

4.1.2 Lack of Automated Tools

The process of mounting a ROP attack can be very tedious and time consuming as explained in the previous section. Currently there is a lack of tools which can aid in automating the classification, selection and generation of the final payload, while maintaining everything architecture-independent.

4.2 Objectives

The preceding section underlines the issues with the existing tools, or lack of them. Despite the existence of some projects which aid in the creation of a ROP attack there is a need for full-fledged architecture-independent solution.

This project aims to fill the need for a dedicated solution by providing the following functionality:

- Architecture-independent Gadget Classification using ESIL[1]

[1]Evaluable String Intermediate Language, used as the IR for Radare2

- ESIL to SMTLib Format Parser
- SMT Generated Payload

4.2.1 Gadget Classification

Gadget Classification Module with the following functionality:

- Assigns a type to each gadget
- Separates effects and side effects
- Provides semantic search for gadgets based on effects
- Stores gadgets for fast retrieval in SDB[1]
- Save/Load functionality for gadgets in SDB across different instances

4.2.2 ESIL to SMTLib2 Format Parser

SMTLib2[2] provides a rigorous standard for describing the input/output language for SMT solvers and its semantics. Providing a module for translating ESIL to SMTLib2 format ensures the flexibility to use any SMT solver which respects the standard.

4.2.3 SMT Generated Payload

For the final ROP payload an SMT solver is used to create it. Having the gadgets translated into SMTLib format the SMT solver requires an end-state formula for which to check the satisfiability. The end state formula represents the goal of the attack, and is provided by the user. Once the formula is satisfied the final module will generate a payload based on the gadgets used by the solver to satisfy the formula, thus facilitating the ROP attack.

[1]String Data Base, SDB represents the internal data base used by Radare2, https://github.com/radare/sdb
[2]http://smtlib.cs.uiowa.edu/

Chapter 5

Radare2 Framework

The Radare[7][8][2] project started in 2006 as a forensics tool, built around a command line hexadecimal editor. Its goal was to provide a free and open source tool to search and recover data from hard-disks. With the open source community support, the project evolved and has grown into a complete framework for reverse engineering, exploiting, fuzzing, binary and data analysis.

In 2009 the framework was completely rewritten due to initial design limitations, thus the current Radare2[3][3] framework was born and continues to grow ever since with the support of the open source community.

5.1 Framework Presentation

At its core the Radare2 framework is still a hexadecimal editor. However the full framework now contains also an assembler / disassembler, code / data analysis tools, graphing tools, scripting features and internal database among others.

One of the main advantages[4] Radare2 has over other reverse engineering tools or frameworks is its versatility, currently supporting over 30 different architectures and instruction sets[5] as well as over 15 different file formats.

All of these have led to the adoption of Radare2 in academia as well as software security researchers for forensics and analysis purposes, Capture The Flag teams across the globe and security-oriented personnel in general. Some of the main use cases include:

- Static Analysis
- Dynamic Analysis

[2] https://github.com/radare/radare
[3] https://github.com/radare/radare2
[4] Radare2 Comparison Table http://rada.re/r/cmp.html
[5] https://en.wikipedia.org/wiki/Radare2#Supported_architectures.2Fformats

- Software exploitation

The framework also encourages a hands-on approach:

> "There is no formal documentation of r2 yet. Not all commands are compatible with radare1, so the best way to learn how to do stuff in r2 is by reading the examples from the web and appending '?' to every command you are interested in."

5.2 Architectural Overview

The Radare2 framework consists of multiple command-line utilities:

- rabin2: tool for extracting information from executable binaries
- rasm2: the assembler / disassembler with multi-architecture support
- rahash2: hashing tool, supports different algorithms: MD4, MD5, CRC16, CRC32, SHA1, SHA256, SHA384, SHA512, etc.
- radiff2: binary diffing tool
- rafind2: pattern-search utility
- ragg2: compiler for high-level language
- rarun2: launcher for running different environments
- rax2: mathematic expression evaluator
- radare2: the core hexadecimal editor and debugger, integrates all of the above

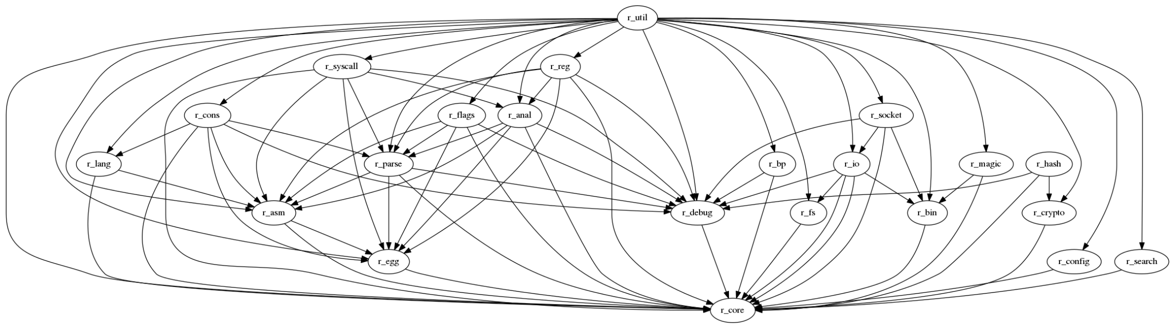


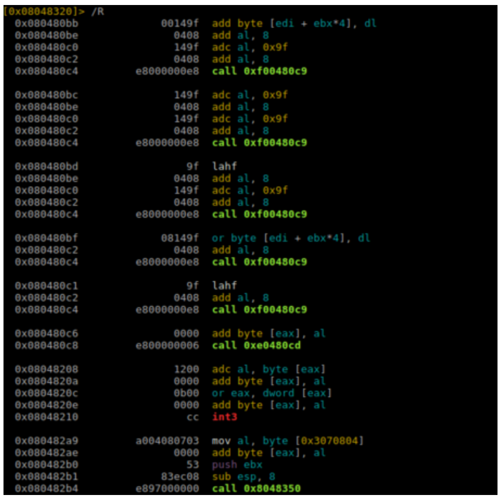
Figure 5.1: Radare2 Module Architecture

5.3 Key Features

There are several specific key features and tools which Radare2 provides that make it the ideal starting point for this project.

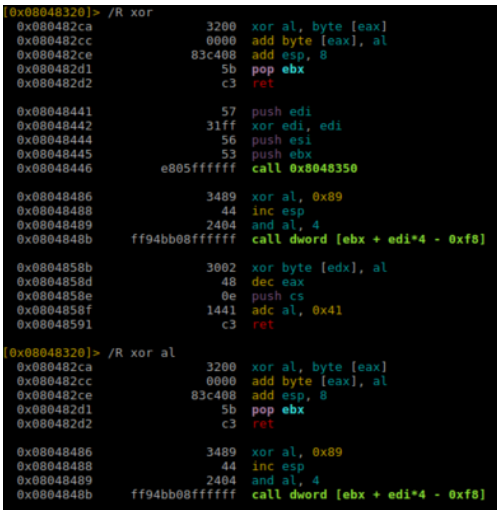
5.3.1 Gadget Finder

The framework already includes a tool for ROP gadget searching (Figure 5.2). This comes with integrated support for Regular Expression matching as well as JSON formatting options for portability.



```
[0x08048320]> /R
  0x080480bb           00149f  add byte [edi + ebx*4], dl
  0x080480be             0408  add al, 8
  0x080480c0             149f  adc al, 0x9f
  0x080480c2             0408  add al, 8
  0x080480c4       e8000000e8  call 0xf00480c9

  0x080480bc             149f  adc al, 0x9f
  0x080480be             0408  add al, 8
  0x080480c0             149f  adc al, 0x9f
  0x080480c2             0408  add al, 8
  0x080480c4       e8000000e8  call 0xf00480c9

  0x080480bd               9f  lahf
  0x080480be             0408  add al, 8
  0x080480c0             149f  adc al, 0x9f
  0x080480c2             0408  add al, 8
  0x080480c4       e8000000e8  call 0xf00480c9

  0x080480bf           08149f  or byte [edi + ebx*4], dl
  0x080480c2             0408  add al, 8
  0x080480c4       e8000000e8  call 0xf00480c9

  0x080480c1               9f  lahf
  0x080480c2             0408  add al, 8
  0x080480c4       e8000000e8  call 0xf00480c9

  0x080480c6             0000  add byte [eax], al
  0x080480c8       e800000006  call 0xe0480cd

  0x08048208             1200  adc al, byte [eax]
  0x0804820a             0000  add byte [eax], al
  0x0804820c             0b00  or eax, dword [eax]
  0x0804820e             0000  add byte [eax], al
  0x08048210               cc  int3

  0x080482a9       a004080703  mov al, byte [0x3070804]
  0x080482ae             0000  add byte [eax], al
  0x080482b0               53  push ebx
  0x080482b1           83ec08  sub esp, 8
  0x080482b4       e897000000  call 0x8048350
```

(a) Basic search



```
[0x08048320]> /R xor
  0x080482ca                3200  xor al, byte [eax]
  0x080482cc                0000  add byte [eax], al
  0x080482ce              83c408  add esp, 8
  0x080482d1                  5b  pop ebx
  0x080482d2                  c3  ret

  0x08048441                  57  push edi
  0x08048442                31ff  xor edi, edi
  0x08048444                  56  push esi
  0x08048445                  53  push ebx
  0x08048446          e805ffffff  call 0x8048350

  0x08048486                3489  xor al, 0x89
  0x08048488                  44  inc esp
  0x08048489                2404  and al, 4
  0x0804848b      ff94bb08ffffff  call dword [ebx + edi*4 - 0xf8]

  0x0804858b                3002  xor byte [edx], al
  0x0804858d                  48  dec eax
  0x0804858e                  0e  push cs
  0x0804858f                1441  adc al, 0x41
  0x08048591                  c3  ret

[0x08048320]> /R xor al
  0x080482ca                3200  xor al, byte [eax]
  0x080482cc                0000  add byte [eax], al
  0x080482ce              83c408  add esp, 8
  0x080482d1                  5b  pop ebx
  0x080482d2                  c3  ret

  0x08048486                3489  xor al, 0x89
  0x08048488                  44  inc esp
  0x08048489                2404  and al, 4
  0x0804848b      ff94bb08ffffff  call dword [ebx + edi*4 - 0xf8]
```

(b) Regex search

Figure 5.2: ROP Gadget Search

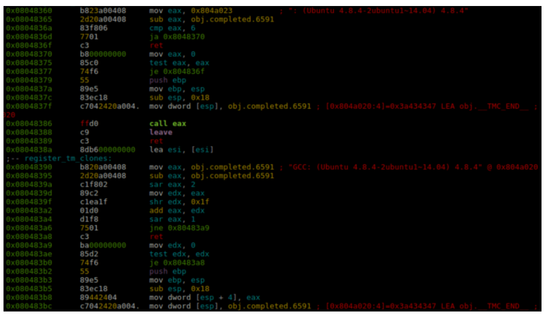
5.3.2 ESIL

ESIL stands for “Evaluable String Intermediate Language” and can be used as an Intermediate Representation for Radare2. It provides a simple representation for every supported instruction set opcode and represents it as a combination of common operations performed by CPUs: binary arithmetic operations, memory loads and stores, syscalls etc. The complete ESIL instruction set aims for simplicity and robustness, a full description can be found in the Radare2 Book[3].

By setting only one environment variable we can for example change the output of the disassembler from assembly to ESIL as seen in Figure 5.3.

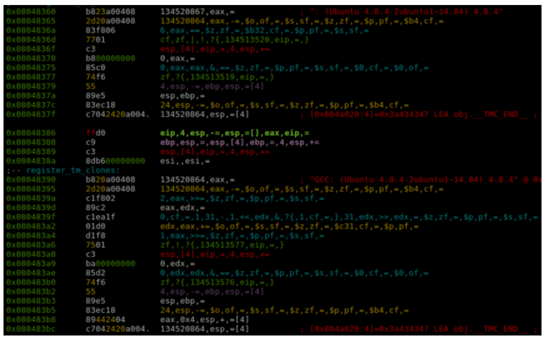
5.3.3 ESIL VM

Radare2 also incorporates a virtual machine for emulating ESIL instructions. The ESIL VM has internal state flags and registers mirroring a real CPU, each emulated instruction having the same effects in the VM as it would on the CPU. This is a crucial component for the gadget classification step as it allows the program to emulate the gadget each instruction at a time and analyze its effects on the machine. Based on the effects each gadget will be assigned into one, or more, categories.



```
0x08048360      b823a00408      mov eax, 0x804a023          ; ": (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"
0x08048365      2d20a00408      sub eax, obj.completed.6591
0x0804836a      83f806          cmp eax, 6
0x0804836d      7701            ja 0x8048370
0x0804836f      c3              ret
0x08048370      b800000000      mov eax, 0
0x08048375      85c0            test eax, eax
0x08048377      74f6            je 0x804836f
0x08048379      55              push ebp
0x0804837a      89e5            mov ebp, esp
0x0804837c      83ec18          sub esp, 0x18
0x0804837f      c7042420a004.   mov dword [esp], obj.completed.6591 ; [0x804a020:4]=0x3a434347 LEA obj.__TMC_END__ ;
020
0x08048386      ffd0            call eax
0x08048388      c9              leave
0x08048389      c3              ret
0x0804838a      8db600000000    lea esi, [esi]
;-- register_tm_clones:
0x08048390      b820a00408      mov eax, obj.completed.6591 ; "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4" @ 0x804a020
0x08048395      2d20a00408      sub eax, obj.completed.6591
0x0804839a      c1f802          sar eax, 2
0x0804839d      89c2            mov edx, eax
0x0804839f      c1ea1f          shr edx, 0x1f
0x080483a2      01d0            add eax, edx
0x080483a4      d1f8            sar eax, 1
0x080483a6      7501            jne 0x80483a9
0x080483a8      c3              ret
0x080483a9      ba00000000      mov edx, 0
0x080483ae      85d2            test edx, edx
0x080483b0      74f6            je 0x80483a8
0x080483b2      55              push ebp
0x080483b3      89e5            mov ebp, esp
0x080483b5      83ec18          sub esp, 0x18
0x080483b8      89442404        mov dword [esp + 4], eax
0x080483bc      c7042420a004.   mov dword [esp], obj.completed.6591 ; [0x804a020:4]=0x3a434347 LEA obj.__TMC_END__ ;
```

(a) Disassemble without ESIL



```
0x08048360      b823a00408      134520867,eax,=                ; ": (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"
0x08048365      2d20a00408      134520864,eax,-=,$o,of,=,$s,sf,=,$z,zf,=,$p,pf,=,$b4,cf,=
0x0804836a      83f806          6,eax,==,$z,zf,=,$b32,cf,=,$p,pf,=,$s,sf,=
0x0804836d      7701            cf,zf,|,!,?{,134513520,eip,=,}
0x0804836f      c3              esp,[4],eip,=,4,esp,+=
0x08048370      b800000000      0,eax,=
0x08048375      85c0            0,eax,eax,&,==,$z,zf,=,$p,pf,=,$s,sf,=,$0,cf,=,$0,of,=
0x08048377      74f6            zf,?{,134513519,eip,=,}
0x08048379      55              4,esp,-=,ebp,esp,=[4]
0x0804837a      89e5            esp,ebp,=
0x0804837c      83ec18          24,esp,-=,$o,of,=,$s,sf,=,$z,zf,=,$p,pf,=,$b4,cf,=
0x0804837f      c7042420a004.   134520864,esp,=[4]             ; [0x804a020:4]=0x3a434347 LEA obj.__TMC_END__ ;

0x08048386      ffd0            eip,4,esp,-=,esp,=[],eax,eip,=
0x08048388      c9              ebp,esp,=,esp,[4],ebp,=,4,esp,+=
0x08048389      c3              esp,[4],eip,=,4,esp,+=
0x0804838a      8db600000000    esi,,esi,=
;-- register_tm_clones:
0x08048390      b820a00408      134520864,eax,=                ; "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4" @ 0x
0x08048395      2d20a00408      134520864,eax,-=,$o,of,=,$s,sf,=,$z,zf,=,$p,pf,=,$b4,cf,=
0x0804839a      c1f802          2,eax,>>=,$z,zf,=,$p,pf,=,$s,sf,=
0x0804839d      89c2            eax,edx,=
0x0804839f      c1ea1f          0,cf,=,1,31,-,1,<<,edx,&,?{,1,cf,=,},31,edx,>>,edx,=,$z,zf,=,$p,pf,=,$s,sf,=
0x080483a2      01d0            edx,eax,+=,$o,of,=,$s,sf,=,$z,zf,=,$c31,cf,=,$p,pf,=
0x080483a4      d1f8            1,eax,>>=,$z,zf,=,$p,pf,=,$s,sf,=
0x080483a6      7501            zf,!,?{,134513577,eip,=,}
0x080483a8      c3              esp,[4],eip,=,4,esp,+=
0x080483a9      ba00000000      0,edx,=
0x080483ae      85d2            0,edx,edx,&,==,$z,zf,=,$p,pf,=,$s,sf,=,$0,cf,=,$0,of,=
0x080483b0      74f6            zf,?{,134513576,eip,=,}
0x080483b2      55              4,esp,-=,ebp,esp,=[4]
0x080483b3      89e5            esp,ebp,=
0x080483b5      83ec18          24,esp,-=,$o,of,=,$s,sf,=,$z,zf,=,$p,pf,=,$b4,cf,=
0x080483b8      89442404        eax,0x4,esp,+,=[4]
0x080483bc      c7042420a004.   134520864,esp,=[4]             ; [0x804a020:4]=0x3a434347 LEA obj.__TMC_END__ ;
```

(b) Disassemble with ESIL

Figure 5.3: Assembly to ESIL

5.3.4 API Hooks

An important aspect of emulation is the ability to setup user-defined hooks in the parser, thus allowing custom analysis or behavior to be implemented for the desired instructions without having to modify the parser each time. This works by calling calling the hook function each time an operation is about to be executed.

By default the ESIL VM implements for example hooks like hook_flag_read(), hook_execute(), hook_reg_read(), hook_reg_write() etc., for logging information such as when a register or flag have been read or written, or when a memory address has been accessed. This information is heavily used in the gadget classification as well.

5.4 Summarize

To sum up, I decided to use the Radare2 Framework for my project because it provides the following tools and features:

- It contains a mature ROP gadget searching tool
- Provides an Intermediate Language and Virtual Machine for emulation and analysis
- Fully cross platform thanks to ESIL VM
- Open source code
- Continuously improving, with new Git commits each day
- Active and friendly community

Chapter 6

Architectural Overview

In the previous chapter the Radare2 Framework was presented, providing an overview of it. The following sections present the new modules introduced as well as how they interact with the framework. A diagram of the architecture can be seen in Figure 6.1.

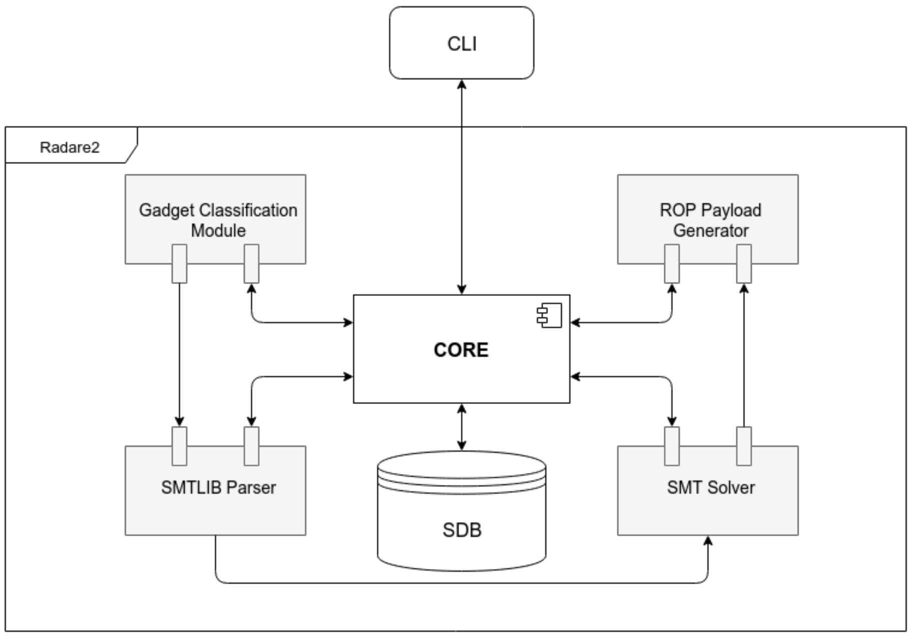


Figure 6.1: Architectural Overview

As can be seen in Figure 6.1 there are four new modules introduced:

- Gadget Classification Module
- SMTLib2 Parser
- SMT Solver
- ROP Payload Generator

In Figure 6.2 we can see a pipeline representation of the modules. This illustrates in more detail how the data flows through the system from the first step of feeding the executable to the system, until the last one when the payload is generated. In the figure we can also notice that each module processes the data and passes the result not only to the next module, but also to a specific namespace in the SDB, corresponding to each module, for storage.

Another important aspect is the SMT Solver Module, in the diagram we can see that this is composed of two logical parts, the actual SMT Solver which is a standalone entity, and a wrapper. The SMT Wrapper is the one responsible with the generation of possible payloads. It in turn uses the SMT Solver to test these payloads until it finds a satisfiable one.

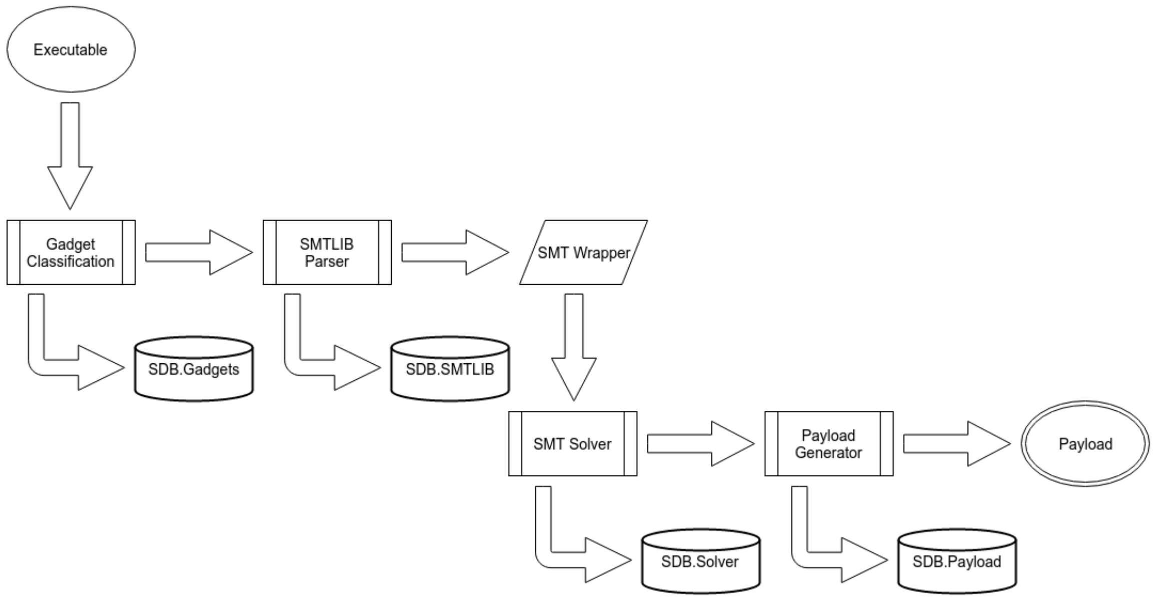


Figure 6.2: Work Flow Diagram

6.1 Radare2 Core

The Radare2 Core represents a key component of the project as it provides access to all the tools and features already implemented in the framework. One of the main features provided by the Core is the ROP gadget search tool as well as access to the SDB and interfacing all modules.

Note that each module or tool can be called individually and in any combination provided the prerequisites for them are respected.

6.2 Gadget Classification Module

This module provides the semantic analysis and classification for each gadget. The raw gadgets are generated by the ROP search feature already implemented in the Core and fed to the module. Each gadget is classified into one or more categories as described in Table 3.1.

6.3 SMTLib2 Translation Module

The SMTLib2 Translation Module implements a parser for translating ESIL to the SMTLib2 format[1]. This module takes the output from the previous classification module and translates it. By introducing this translation module for SMTLib2 format between the classification step and actual SMT Solver, the user has the possibility to plug in any SMT Solver which uses this standard, instead of having to individually translate the input for custom solvers.

6.4 Payload Generation Module

If the SMT Solver finds the formula satisfiable it generates a payload for the specific executable in question. The module determines which gadgets were used to satisfy the target formula and generates a payload with their corresponding addresses.

6.5 SDB

Each of the previously described modules use the String Data Base present in Radare2 to store their results. The modules first check the SDB to see if the same task was already processed and perform a quick lookup before processing it. The user can query the results of each individual step and inspect or modify it, as well as save and restore the results into projects and use them across different sessions.

Chapter 7

Implementation Details

Taking into account the style of the Radare2 framework, our project is written in pure C. All data structures used are custom defined for each task.

7.1 Gadget Classification

For the classification stage, the sequence of assembly instructions which represent gadgets are translated into the ESIL equivalent and stored as a list. For each type of gadget there is an independent function which checks if the gadget fits that specific type. The generalized algorithm can be seen in Algorithm 1.

Algorithm 1: Generic Classification Algorithm

```
Data: Gadget: List of ESIL instructions
Result: Classified Gadget
1 clone VM state;
2 for each ESIL instruction do
3     emulate instruction;
4     inspect VM state;
5     if postcondition == TRUE then
6         assign gadget type;
7 restore VM state;
```

The API Hooks present in the ESIL VM are used for checking the constraints and postconditions in order to determine if and how registers, flags or memory were changed.

Each classification function has a different set of constraints and postconditions it checks. For example a NOP (No Operation) gadget is the easiest to classify, we can inspect the information returned by the hooks, if there is none (meaning no changes whatsoever occurred), it is clearly a NOP. However there can be gadgets such as *xchg eax, eax* which will log a register read and write from the API Hooks, but it will still be classified as a NOP since semantically it changes nothing.

Other patterns, such as StoreMem will first of all check if there was a MEM write, if not all other information is ignored and it jumps to the next pattern for classification.

7.2 ESIL to SMTLib2 Parser

The Parser Module takes the ESIL version of the classified gadgets and generates the SMTLib2 equivalent. A generalized algorithm can be seen in Algorithm 2.

Algorithm 2: Generic Translation Algorithm

```
Data: ESIL instruction
Result: SMTLib2 formula
1  extract_type(instruction);
2  operator = type;
3  oprnd_lst = [];
4  for each operand in instruction do
5      if operand == REGISTER then
6          opr = translate_register(operand);
7      else
8          opr = translate_immediate(operand);
9      oprnd_lst.append(opr);
10 check_constraints(operator, oprnd_list);
11 generate_expression(operator, oprnd_lst);
```

Each instruction type (add, sub, xor etc.) has a specific functions which handles the translation and constraint checking.

Since modern CPUs use arithmetic over fixed-size bit vectors it seemed only natural to represent the registers as bit vectors for the SMT solver. The SMTLib2 standard supports bit vectors of arbitrary size and provides a large number of functions over them, thus allowing the encoding of all instructions through them.

An example of the transition from assembly to ESIL to SMTLib2 can be see in Table 7.1.

7.3 SMT Solver

Due to the SMTLib2 Parser the user can plug in any SMT Solver which respects the standard, offering the choice and flexibility to change it as he/she sees fit. For the actual SMT Solver we decided to use an open source solver, Z3[1].

The SMT Module represents the final step in the payload generation. Based on the end state formula which represents the goal of the attack, the module tries to find a combination of gadgets for which the formula is satisfiable. The first attempt consisted of doing an exhaustive search of all combinations possible of k-length, where k is an arbitrary number.

[1]Note that ESIL instructions also contain a part for flag changes, they were omitted in this example for clarity

[1]https://github.com/Z3Prover/z3

x86 Assembly	**ESIL instruction**	**SMTLib2 expression**
xor eax, ebx	ebx, eax, ^=	(= eax (bvxor eax ebx)
add esp, 8	8, esp, +=	(= esp (bvadd esp #x00000008)
mov ecx, edx	edx, ecx, =	(= edx ecx)
mov eax, 0x80085	0x80085, eax, =	(= #x00080085 eax)
push edi	4, esp, -=, edi, esp, =[4]	(= esp (bvsub esp #x00000004)) (= (concat (select MEM (bvadd edi #x00000003)) (select MEM (bvadd edi #x00000002)) (select MEM (bvadd edi #x00000001)) (select MEM (bvadd edi #x00000000))) esp)

Table 7.1: Assembly to ESIL to SMTLib2 Translation Example[1]

This approach of trying all combinations of k-length and checking the satisfiability works when applied on small executables with simple constraints. By small we understand an executables which have a number of gadgets between 100-200. These represent hello-world like programs and simple vulnerable programs such as Listing 8.1. For simple end state formulas a $k = 10$ will be enough. The main issue with this approach is that it has a O(N^k) complexity where N is the number of gadgets. This can get quickly out of hand, for example the */bin/ls* on Linux Ubuntu x86_64 has aprox. 12500 gadgets, leading to a dramatic increase in computing time. Although this approach is correct and works, it requires long computing times for executables with more than a few hundred gadgets thus making it virtually unusable.

7.3.1 Optimizations

To make the payload generation usable the process needs to be optimized from this initial approach. For this I came up with the following optimizations:

7.3.2 Reduced Search Space

The initial approach used all possible gadgets found by Radare2 without discrimination, this includes gadgets ending in the *call* or *jmp* instruction. A first optimization was to reduce the number of gadgets, thus I decided to use only the gadgets ending in the *ret* instruction. This lead to dramatic decrease of the search space. The */bin/ls* which initially contained aprox. 12500 general gadgets can be reduced to only 1500 ret-gadgets, leading to an 88% decrease in the number of gadgets.

7.3.3 Utility Metric

While reducing the number of gadgets used reduces the time it takes to find a satisfiable payload, if it exists, it can still take up a considerable time when searching for longer ROP payload with a length bigger than 10. We needed a way of telling the SMT Solver which gadgets would be preferable to use over others. To implement this behavior of favoring certain gadgets I introduced an utility metric. The concept behind it is simple, always favor the gadgets with minimal side effects and use those when possible. Each gadget gets assigned an utility value based on how many side effects it contains, clobbering registers is considered the most severe one, while clobbering flags or memory accesses are considered less severe.

The utility metric optimization can however have a negative aspect if certain constraints are not set in place. Since a gadget can be reused for an unlimited number of times, it may reuse the same ones, neglecting other gadgets with lower utility metric that might lead to a solution.

Chapter 8

Testing and Evaluation

The testing and evaluation was conducted using an Ubuntu 16.04 x86_64 system, primarily on 32 bit custom made executables. To test the correctness of the program we feed it an executable for which we know the correct results for the classification and translation step, and cross check that they match.

8.1 Testing Approach

Due to Radare2's volatile nature, with new commits and PRs being merged each day, a solid testing infrastructure is required to assure nothing breaks and functionality is preserved after each new patch.

Radare2 already implements a dedicated separate repository[2] for testing, containing a few hundred tests. This repository has been enhanced with my own tests for the ROP payload generation.

8.1.1 Unit Testing

To ensure the correct functionality of the newly introduced modules and features unit tests have been introduced to check the classification and translation features work as expected. For testing a small set of 10-15kb sized executables are used, these are represented by hello-world like programs and other simple ones which cumulative include all types of gadgets.

8.1.2 Regression Testing

During the development of this project there have been situations in which the behavior or functionality of a feature has been disrupted or broken completely by a new patch. To prevent

[2]https://github.com/radare/radare2-regressions

this regression tests have been introduced to check that the gadget classification, translation and payload generation modules maintain functionality with each new patch.

8.1.3 System Testing

For a complete end-to-end check we use a couple of system tests. Listing 8.1 shows a simple code used for this. The first test compiles the code as is, classifies the gadgets, translates them and attempts to create a ROP payload. The payload generation will fail due to an incomplete set of gadgets.

For the second test we compile the code with the libc statically linked. This guaranties enough gadgets for a Turing Complete instruction set.

```c
void vulnerability(char* input) {
        char buffer[1024];
        strcpy(buffer, input);
}

int main(int argc, char** argv) {
        vulnerability(argv[1]);
        return 0;
}
```

Listing 8.1: Vulnerable Program

8.2 Real World Testing

Testing the project in a real world scenario is imperative since the goal is full integration with a real world reverse engineering framework used in industry, academia and by professionals in their fields.

For evaluating the efficiency of the classification and translation module we inspected the results after running the modules on several executables and thoroughly inspecting each result.

In the first phase of testing a set of 32 bit rudimentary executables[1] were used as input for the modules. Using small executables allowed us to manually inspect these stages. The results show that for the basic x86 instructions the classification and translation work as expected.

The second phase of testing was conducted on executables from the */bin* directory present on Linux Ubuntu x86_64, among these we can count the *ls, cat, sleep* binaries. Since the executables used in this stage present a more complex design and increased size they normally generate thousands of possible gadgets, thus rendering a manual inspection inefficient and too time consuming to consider. Instead a statistical approach was used, where a number random

[1]Hello-Worlds, Simple Number Addition, Branching Examples, Loop Examples etc. All programs from the rudimentary section consist of 50 or less lines of code

gadgets from different offsets were extracted from each executable and inspected. This showed that for complex 64 bit binaries there are still some discrepancies and issues to tackle for the classification phase.

The following section presents some of the issues encountered.

8.2.1 Encountered Issues

The testing on real world binaries and complex programs with a bigger code base led to the discovery of several issues:

- For binaries which present more than 3000 gadgets the classification phase is not fast enough in its current form, it can take up to a couple of minutes depending on the binary.
- It was discovered that for the x86_64 instruction set ESIL ignores a specific opcode, thus leading to a potential incorrect classification.
- For binaries with more than a few thousand binaries the computing time for the payload generation can grow exponentially based on the length of the payload and the complexity of the end state formula.

These issues are being investigated and will be solved in the near future.

8.3 Community Feedback

Throughout the development of this project I received continuous feedback from the core developers and original author of Radare. They have constantly assisted and aided in the development process and proved to be an invaluable source of information.

I would like to thank Sergi Alvarez, Anton Kochkov and Jeff Crowell for all the help they provided.

The next step will be to gather feedback from the non-developer users of Radare2 and integrate the results in a future patch.

Chapter 9

Conclusions and Further Work

In this thesis I presented a new tool to aid and automate the construction of ROP-based payloads. Based on the Radare2 Framework I introduced four new modules, each with distinct features, into the framework. The current implementation status has met the objectives proposed in Chapter 4.2. However testing and feedback revealed that there issues which need to be addressed before the project can be fully integrated in the next stable release of the framework.

The following sections present the current implementation status as well as new aspects and features which need to be addressed in further work on the project.

9.1 Current Status

The tool can be run on any platform / architecture that supports Radare2 and provides the following features:

- Semantic classification for gadgets
- Translation from ESIL to the SMTLib2 standard
- Automatic payload generation using a SMTLib2 compliant SMT Solver
- Separate caching and querying functionality for each module for fast lookup in SDB
- Save / Load functionality for the results across different instances

Currently the Gadget Classification module is already present in the latest release of Radare2[2].

The SMTLib2 Parser and Payload Generation module require more testing and validation for complex binaries before being integrated with the next release.

[2] https://github.com/radare/radare2/releases/tag/0.10.5

9.2 Further Work

Further work will focus on fixing the main issues discovered from testing and feedback. Afterward independent features and improvement will be considered. The following subsections detail the issues and improvements discovered from feedback.

9.2.1 Improve Performance

As mentioned in Section 8.2.1 there is a significant time slot currently spent in the searching, classification of gadgets and payload generation. This requires further investigation and optimization in order to make the tool viable for use on large, real-life binaries.

9.2.2 Large Scale Testing

Afterwards large scale testing is required to thoroughly analyze the behavior and validate the tool in different environments and on a large variety of binaries. For this task we suggest validating the classification and translation phase on the executable present in the */bin* directory for the major Linux distributions (Ubuntu, Debian, Fedora, RedHat, etc.) on both 32 and 64 bit architectures, as well as on Windows.

For an end-to-end test we propose using different write-ups[1] of Capture The Flag (CTF) competitions which can be found online. These provide an ideal environment for testing as we can cross for example the solution obtained by our tool for a ROP attack with the solution proposed by the write-up and compare both functionality and efficiency.

9.2.3 Cross Platform Testing

Due to the fact that the classification modules uses an intermediate representation language, ESIL, as input, the whole architecture independent attribute of the tool is guarantied by the Radare2 Framework itself.

However a test suite for different architectures is required to validate this. Currently the vast majority of tests are only on Intel's x86 and x86_64 architectures.

9.2.4 Graphical User Interface

At present the tool is accessible only through a Command Line Interface, in the same spirit with Radare's CLI approach. However this adds to the already difficult learning curve the framework presents.

Taking into account the feedback already received from the community a Graphical User Interface will be a main topic for future development.

[1]Example of write-ups which can be found online https://github.com/ctfs/

9.2.5 Constraint Generation Manager

Another key item which kept coming up in the feedback is the current way of setting the target end state for the payload generator. At the moment the user is required to encode the final target state as a sequence of formulas for the SMT Solver which represent the constraints (register's and memory state) which need to be satisfied by the payload.

A simple, intuitive manager for generating these constraints based on a natural language is required. The user should be allowed for example to specify for the end state that he wishes to spawn a shell or a certain executable to be called with a specific argument. The constraint manager will generate the SMT formula equivalent for this and pass it to the Payload Generation module. This represents an important aspect for further development.

Bibliography

[1] Clark Barret, Pascal Fontaine, and Cesare Tinelli. The smt-lib standard version 2.5. 2015.

[2] Christian Heitman and Ivan Arce. Barf: A multiplatform open source binary analysis and reverse engineering framework. 2014.

[3] Maxime Morin (maijin). The radare2 book. 2016.

[4] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 4(58), 2001.

[5] Taeho Oh. Advanced buffer overflow exploits. October 1999.

[6] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[7] Sergi Alvarez (pancake). The original radare book. 2008.

[8] Sergi Alvarez (pancake). Manual binary mangling with radare. *Phrack*, November 2009.

[9] Alexander Peslyak(aka Solar Designer). "return-to-libc" attack. *Bugtraq*, August 1997.

[10] Alexander Peslyak(aka Solar Designer). Stackpatch. *Bugtraq*, April 1997.

[11] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM*, 2012.

[12] Jonathan Salwan. Ropgadget tool. 2011.

[13] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. *Usenix Security Symposium*, 2011.

[14] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). *ACM*, 2007.